Rust Book - 5.1: Structures

#rust Defining and Instantiating Structs

A **struct** , or **structure** , is a custom data type that lets you name and package together multiple related values that make up a meaningful group. - a **struct** is like an object's data attributes.

structs are more flexible than tuples: you don't have to rely on the order of the data to specify or access the values of an instance.

`struct` + inside curly brackets, we define the names and types of the pieces of data, which we call **fields** .

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

create an **instance** of that struct

```
let user1 = User {
    email: String::from(“someone@example.com”),
    username: String::from(“someusername123”),
    active: true,
    sign_in_count: 1,
};
```

```
user1.email = String::from(“anotheremail@example.com”);
```

Note that the entire instance must be mutable; Rust doesn't allow us to mark only certain

fields as mutable. As with any expression, we can construct a new instance of the struct as the last expression in the function body to implicitly return that new instance.

```
fn build_user(email: String, username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

Using the Field Init Shorthand when Variables and Fields Have the Same Name

```
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

Creating Instances From Other Instances With Struct Update Syntax

to create a new instance of a struct that uses most of an old instance's values but changes some.

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    active: user1.active,
    sign_in_count: user1.sign_in_count,
};
```

The syntax `..` specifies that the remaining fields not explicitly set should have the same value as the fields in the given instance.

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    ..user1
};
```

Using Tuple Structs without Named Fields to Create Different Types

You can also define structs that look similar to tuples, called tuple structs. Tuple structs have the added meaning the struct name provides but don't have names associated with their fields; rather, they just have the types of the fields. Tuple structs are useful when you want to give the whole tuple a name and make the tuple be a different type than other tuples, and naming each field as in a regular struct would be verbose or redundant.

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);
let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

Note that the `black` and `origin` values are different types, because they're instances of different tuple structs. Each struct you define is its own type, even though the fields within the struct have the same types. Tuple struct instances behave like tuples: you can destructure them into their individual pieces, you can use a `.` followed by the index to access an individual value, and so on.

Unit-Like Structs Without Any Fields

You can also define structs that don't have any fields! These are called unit-like structs because they behave similarly to `()`, the unit type. Unit-like structs can be useful in situations in which you need to implement a trait on some type but don't have any data that you want to store in the type itself.

Ownership of Struct Data

Ownership of Struct Data

In the `User` struct used the owned `String` type rather than the `&str` string slice type. This is a deliberate choice because we want instances of this struct to own all of its data and for that data to be valid for as long as the entire struct is valid.

It's possible for structs to store references to data owned by something else, but to do so requires the use of **lifetimes**. Lifetimes ensure that the data referenced by a `struct` is valid for as long as the `struct` is.

```
struct User {
    username: &str,
    email: &str,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        email: “someone@example.com”,
        username: “someusername123”,
        active: true,
        sign_in_count: 1,
    };
}
```

The compiler will complain that it needs lifetime specifiers:

```
error[E0106]: missing lifetime specifier
 ->
  |
2 |     username: &str,
  |               ^ expected lifetime parameter

error[E0106]: missing lifetime specifier
 ->
  |
3 |     email: &str,
  |            ^ expected lifetime parameter
```

Method Syntax

Method Syntax

Methods are similar to functions: they're declared with the `fn` keyword and their name, they can have parameters and a return value, and they contain some code that is run when they're called from somewhere else.

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Rust doesn't have an equivalent to the->operator; instead, Rust has a feature called *automatic referencing and dereferencing*. Calling methods is one of the few places in Rust that has this behavior.

Here's how it works: when you call a method with `object.something()`, Rust automatically adds in `&`, `&mut`, or `*` so `object` matches the signature of the method. In other words, the following are the same:

```
p1.distance(&p2);
(&p1).distance(&p2);
```

The first one looks much cleaner. This automatic referencing behavior works because methods have a clear receiver—the type of `self`. Given the receiver and name of a method, Rust can figure out definitively whether the method is reading (`&self`), mutating (`&mut self`), or consuming (`self`). The fact that Rust makes borrowing implicit for method receivers is a big part of making ownership ergonomic in practice.

Associated Functions

Associated Functions

Another useful feature of `impl` blocks is that we're allowed to define functions within `impl` blocks that don't take `self` as a parameter. These are called *associated functions* because they're associated with the struct. They're still functions, not methods, because they don't have an instance of the struct to work with.

Associated functions are often used for constructors that will return a new instance of the struct. For example, we could provide an associated function that would have one dimension parameter and use that as both width and height, thus making it easier to create a square `Rectangle` rather than having to specify the same value twice:

Structs let you create custom types that are meaningful for your domain. By using structs, you can keep associated pieces of data connected to each other and name each piece to make your code clear. Methods let you specify the behavior that instances of your structs have, and associated functions let you namespace functionality that is particular to your struct without having an instance available.